Computer Vision Assignment

Michael Simpson

EEECS
Queen's University Belfast
Northern Ireland
2nd February 2025
Word Count: 2049

Introduction

In the past few decades, many models have been developed that enable computers to examine an input image and output meaningful features, something that humans can easily do. Some of these models were constructed as part of the following assignment, as well as their efficacy examined. These models included the SIFT (Scale-Invariant Feature Transform) and ORB (Orientated FAST and rotated BRIEF) models for feature selection and were paired with Linear SVC to classify images. As well, both a regular Neural Network (NN) and a Convolutional Neural Network (CNN) were constructed, and the difference in accuracy was assessed between all models. These models were trained on the TinyImageNet100 dataset which contains 100 classes of images, each containing 500 images. An example of one of these images can be seen below. Each image is similar in terms of pixilation, making this dataset quite difficult for the models to train on and correctly classify test inputs. This difficulty was necessary however, as it allowed for more rigorous testing to be performed on the models.

Figure 1: Example image from the TinyImageNet100 dataset. It was taken from a class containing images of snakes, but it is clearly difficult to tell.

Phase 1: Data Preparation

Figure 2 below shows the file directory to all the images used in this assignment. The classes are referred to, for example, as 'n01443537', with each containing 500 images of a particular type of animal. 15 of these classes were chosen at random meaning each model had to choose between 15 different potential answers when classifying an image, making this quite a challenge for the models. A function was created to retrieve these random classes, with several classes being omitted entirely from the selection. This was because both the SIFT and ORB models struggled to detect any features for certain images, thus resulting in an error. Therefore, the classes containing these images were removed from the selection

process until those models could detect features in all the images. The code shown below was used to omit these classes belonging to a list.

Figure 2: An example of the file directory so all the images within the 100 classes

```
image_dir_SIFT = [x for x in image_dir if x not in remove_SIFT]
```

Figure 3: Code used to remove unwanted classes from the directory. These unwanted classes were stored in a list within the function

The dataset then had to be split up intro training and test data. This was done by creating another function that retrieved the random classes using the function mentioned before. Each class was then iterated through with the first 400 images being added to a training list and the last 100 added to a test list. Each element within the lists was then a directory to each image for ease of access. The target labels for each image were also retrieved within the function as these were used to train the models. An example of part of the training list is shown in Figure 4 for clarification.

Phase 2: Hand-crafted feature approach

The first method of image classification was using the SIFT and ORB models. Both are methods of feature detection, such that for a given image, they are able to identify keypoint descriptors, which are interesting sections of the image. For SIFT, the keypoints are detected using a Difference of Gaussian (DoG) approach, which finds points where the difference in intensity of pixels is large, making it scale-invariant. The keypoint is then described by a 128-dimension descriptor which is obtained from histograms describing the gradient intensities around the keypoint. On the other hand, ORB uses the FAST algorithm that detects pixesls in a circular neighbourhood, and is much faster than DoG. ORB then produces a binary descriptor by comparing the intensity of pairs of pixels.

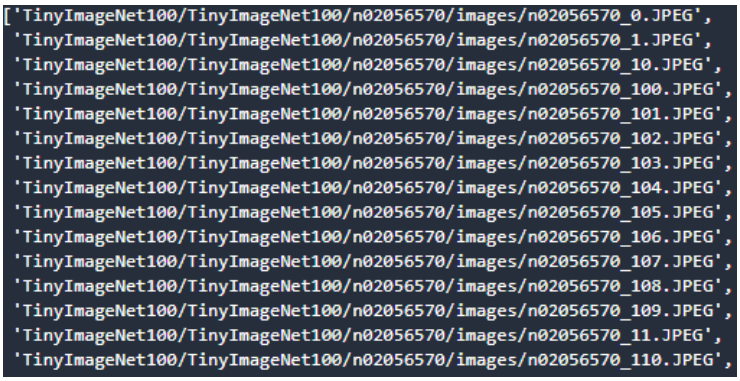

Figure 4: Within the training list, each element is the file directory to an image.

Both of these methods were given their own functions. It should be noted that different classes were used for each model as ORB struggled to find features in the classes used for SIFT, and vice versa. The descriptors were obtained for each image and stored in a matrix variable. Using K-Means clustering, the descriptors were then clustered into a chosen number of clusters (10 in this case), where each cluster centre represented a visual word associated with the image. The centres of these clusters where then returned from the function.

In order to apply the Linear SVC model, the SIFT descriptors had to be transformed in feature vectors for each image. This was done by building a histogram where each bin corresponds to a visual word and the value is how many descriptors were assigned to that bin. Obtaining the feature vector for all images was then used alongside the target labels to train the model and then predict labels using the feature vectors from the test set. The figures below show the results of using Linear SVC on both the SIFT and ORB feature vectors in the form of a classification report and Confusion Matrix. The overall accuracy of SIFT and ORB came out to be 27% and 17%, respectively.

Phase 3: Neural Network

In the context of image analysis, a Neural Network takes in raw pixel data in the form of a 1D vector where weights and activation functions are applied in the hidden layer. This is the forward propagation of the input data. The model then makes a target prediction, of which is compared to the true target value to obtain a loss function. This function is used to inform the model how it should change the weights and biases in order to decrease the loss (this is the derivative of the weights with respect to the loss). The weights are then adjusted so the model can make better predictions of which class the image belongs to. This type of NN is called a fully-connected NN and only involves the input, hidden and output layers.

Class	Precision	Recall	F1-Score
0	0.24	0.29	0.26
1	0.15	0.10	0.12
2	0.21	0.15	0.18
3	0.30	0.66	0.41
4	0.29	0.21	0.24
5	0.20	0.08	0.11
6	0.13	0.03	0.05
7	0.27	0.41	0.33
8	0.24	0.04	0.07
9	0.30	0.24	0.27
10	0.30	0.27	0.29
11	0.22	0.36	0.27
12	0.38	0.64	0.48
13	0.18	0.16	0.17
14	0.3	0.38	0.33
Accuracy			0.27

(a) SIFT Classification Report

Class	Precision	Recall	F1-Score
0	0.17	0.24	0.20
1	0.12	0.04	0.06
2	0.18	0.38	0.24
3	0.00	0.00	0.00
4	0.13	0.40	0.20
5	0.15	0.06	0.09
6	0.21	0.20	0.21
7	0.21	0.12	0.15
8	0.18	0.06	0.09
9	0.18	0.52	0.27
10	0.22	0.02	0.04
11	0.04	0.01	0.02
12	0.18	0.38	0.25
13	0.00	0.00	0.00
14	0.31	0.18	0.23
Accuracy			0.17

(b) ORB Classification Report

Table 1: 15 class Classification Reports for both SIFT and ORB models. Precision, recall and f1-score are given for each class as wel as the overall accuracy of each model

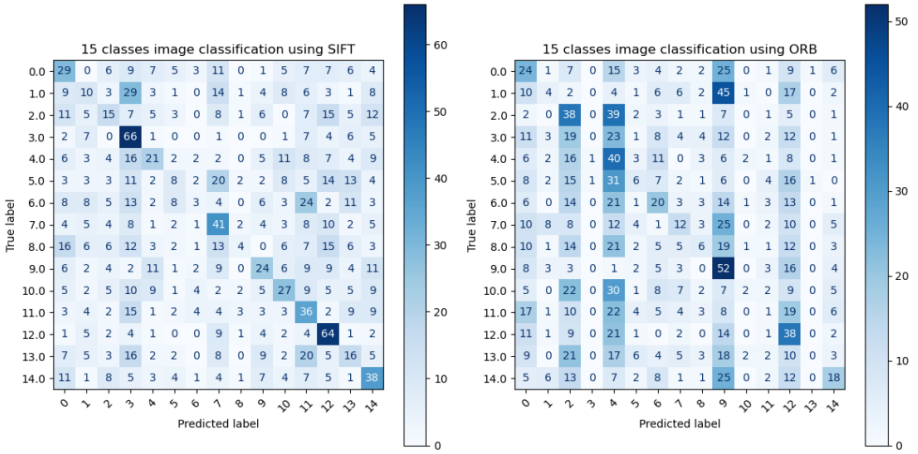

Figure 5: Confusion Matrices for both SIFT (left) and ORB (right) models when applied to 15 class Classification

In order to pass the images through the NN, they first had to be pre-processed. This was done by transforming each image to tensor format, which

is necessary for the model to perform an analysis. All tensors were then normalized and sorted into batches using a Dataloader which contained all the image class labels as well.

The NN itself contained a sequential layer that performed certain operations on the input data. This included a flattening of the image into an array containing 12288 elements (64 x 64 x 3), then a dropout function which introduced a probability that a neuron would be zeroed, thus increasing the model's ability to generalise. As well, a ReLU activation function to introduce non-linearity and finally a Softmax function, necessary for multi-class classification.

During the training of the model, some parameters had to be changed to increase the accuracy including the learning rate and epoch number, which was the number of times the model would train on the dataset. For each epoch, the model trained on both the training and test datasets, where the weights were not updated for the test set. This allowed for a comparison of training and validation accuracies. If the training accuracy was much higher than the validation accuracy, then the model was overfitting the data and parameters and hyperparameters had to be modified to change this. As well, some code was included to cease training of the model when training accuracy strayed too far from the validation accuracy. This code is shown below as well as the resulting graph of training and validation accuracy against epoch number.

```
if test_loss_list[epoch] < best_val_loss:
    best_val_loss = test_loss_list[epoch]
    early_stop_counter = 0  # Reset the counter if validation loss improves
else:
    early_stop_counter += 1  # Increment the counter if no improvement

# Check for early stopping
if early_stop_counter >= early_stop_patience:
    print(f"Early stopping triggered at epoch {epoch}")
    break
```

Figure 6: This code was used to implement a stop for the model when the training accuracy strayed too far from the validation accuracy. It counts when the validation accuracy does not improve and the model stops when this counts reaches a chosen value

After running of 150 epochs, the trained model resulted in the final accuracy and loss found in the table below, for both the training and test sets.

	Training	Test
Accuracy	30.4%	26.8%
Loss	2.53	2.55

Table 2: The final accuracies and losses for both the training and test sets for the fully-connected NN

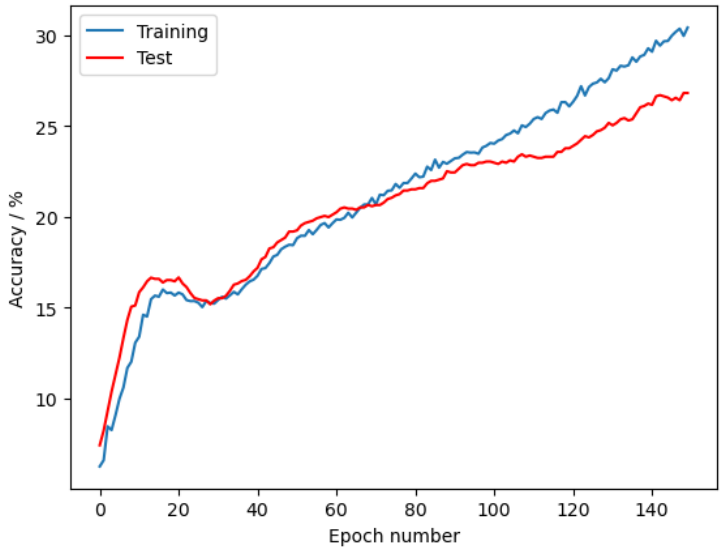

Figure 7: Graph showing the change in Training and Test accuracy over epoch number using the fully-connect Neural Network.

Phase 4: Convolutional Neural Network

A CNN works in a similar way to a fully-connected NN but has extra steps being the convolutional layers that allow for more powerful image analysis. This starts with a kernel that usually has smaller dimensions than the input image and moves over the image extracting mid-level features such as edges. The kernel size can be specified as well as the size of the steps it takes when moving over the image (stride), and the padding which adds a layer of empty pixels around the image. Another layer involves pooling which further reduces the dimensionality of the convolved layer by extracting dominant features that are positional and rotational invariant. The CNN can also include other layers similar to the fully-connected NN such as activation and dropout functions. The output of the CNN is then passed through a fully-connected layer to classify the input image.

The CNN designed for this assignment contains two blocks and a classifier. The first block contains two convolutional layers to filter important features out of the image, and a normalization layer and activation function in between. The normalization layer normalizes activations across the batches to improve stability and reduce overfitting. At the end of the first block is a Max Pooling layer that downsizes the feature map even more and extracts important spatial information. The second block then performs more convolution and pooling before passing the feature map to the classifier layer where it is flattened to a 1D tensor. After this, the same functions were used to make class predictions and calculate the overall loss and accuracy of the model.

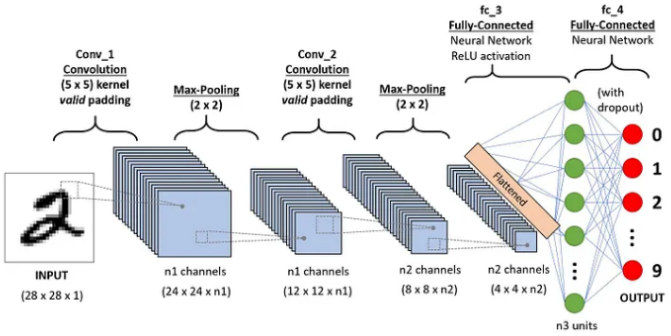

Figure 8: Example CNN to classify handwritten digits [1]

The input data was also pre-processed before passing through the model. The training data was normalized in the same way it was for the fully-connected NN, but other transforms were performed as well, including random rotation and horizontal flip, random affine to affect the shape of the image, and also random color jitter to change the certain properties such as hue. All of this allowed for the model to learn deeper patterns in the images and generalize to a greater extent. The simulation ran for 15 epochs and produced the results shown in the table below.

	Training	Test
Accuracy	42.3%	42.0%
Loss	1.80	1.87

Table 3: The final accuracies and losses for both the training and test sets for the Convolutional NN

Discussion and interpretation

The accuracies obtained for SIFT and ORB shown in Table 1 indicate a low performance for both models. This is probably due to the low quality of the images, as evidenced by Figure 1. Combining this with the fact that SIFT and ORB perform by detecting important features from the images directly, the low performance makes sense, as it is very unclear where these features are, even to the human eye. Another possible reason for this low performance could be due to the fact that only RGB images were kept for the analysis, so the size of the dataset was slightly reduced, and thus less data could be trained on.

The difference in accuracy between SIFT and ORB can also be explained by the fact that SIFT performs floating-point operations and produces large de-

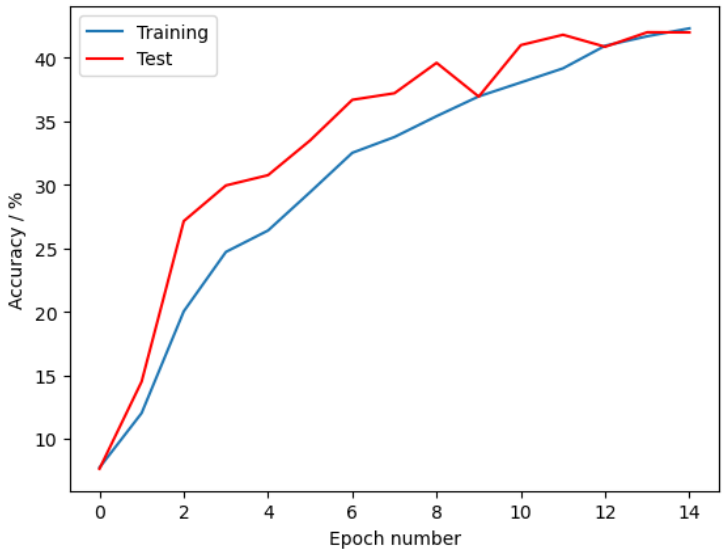

Figure 9: Graph showing the change in Training and Test accuracy over epoch number using the Convolutional NN

scriptors, which is much more useful for analysing these images. ORB performs alot faster but fails to capture the important details in the images to make accurate predictions.

It is interesting that the SIFT accuracy was around the same as the test accuracy found using the fully-connected NN, as one might expect a NN to perform much better. But this is not necessarily the case, as the input for the NN was only a 1D vector of pixel values, making it difficult for the model to capture important features and gain spatial awareness of an image. SIFT, on the other hand, looks at the image as a whole and extracts important information. If the images were more clear and dataset was larger, then perhaps the fully-connected NN might have performed better than the SIFT model.

It is clear the CNN performs the best out of all the models. This is clearly due to the model's ability to convolve the images before passing through the fully-connected layer, as this extracts many of the important features of the images. However, the final accuracy is still not very high, as it shows that the model is correctly predicting the class of the image just over a third of the time. This is only for 15 epochs as compared to 150 for the fully-connected NN, but anymore epochs for the CNN would have been too computationally expensive. Many different parameters were changed and introduced to try and increase the accuracy such as a dropout rate and altering the kernel size but these did not appear to help. The architecture of the CNN itself may have to be drastically altered to further increase the accuracy, but for now, the CNN can be deemed

somewhat successful as it was capable of predicting classes better than the other models.

Conclusion

This project has been an investigation of 3 different models and their application to image classification. The dataset given was the TinyImageNet100, containing 100 classes each with 500 images. It was found that the fully-connected NN and hand-crafted feature approach (SIFT and ORB), performed similarly, whereas the CNN performed best out of the 3. This was expected as it is generally agreed that CNNs are the best models for image analysis. The model could be improved, however, with further alterations to it's architecture and perhaps more training time.

Appendix

```
#Import relevant libraries
import os
import random
import numpy as np
import cv2
from skimage.feature import SIFT, ORB, fisher_vector,
    learn_gmm
from skimage.transform import resize

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report,
    ConfusionMatrixDisplay

from PIL import Image

import torch
from torchvision import transforms
from torch.utils.data import TensorDataset, DataLoader
from torch import nn

from helper_functions import accuracy_fn

import requests
from pathlib import Path
```

```
from timeit import default_timer as timer

from tqdm.auto import tqdm

# Download helper functions from Learn PyTorch repo (if
    not already downloaded)
if Path("helper_functions.py").is_file():
  print("helper_functions.py already exists, skipping
      download")
else:
  print("Downloading helper_functions.py")
  # Note: you need the "raw" GitHub URL for this to work
  request = requests.get("https://raw.githubusercontent.
      com/mrdbourke/pytorch-deep-learning/main/
      helper_functions.py")
  with open("helper_functions.py", "wb") as f:
    f.write(request.content)

#Function to pick out random classes from the dataset
def get_classes(path, num):

    image_dir = os.listdir(path) #Path directory to
        classes

    #Classes to be skipped as they contain images that
        SIFT and ORB cannot process as they are
    remove_SIFT = ['n02190166', 'n01784675','n01910747','
        n02231487','n02233338','n02236044',
            'n02113799','n02124075','n02125311','
                n02321529','n01917289','n02669723',
            'n02815834','n01855672','n02165456','
                n01945685','n01950731','n02791270','
                n02892201','n02906734','n02909870','
                n02058221'
            ,'n03026506','n01644900','n02085620','
                n02085620','n02423022','n01698640']

    remove_ORB = ['n02321529','n01698640', 'n02909870', '
        n02906734', 'n03126707','n0232152','n02480495','
        n02233338','n01910747',
                'n03160309', 'n02730930','n01945685','
                    n02788148','n02226429','n02437312','
                    n03026506','n03201208','n02999410','
                    n01882714',
                'n02963159', 'n03250847']
```

```
    image_dir_SIFT = [x for x in image_dir if x not in
        remove_SIFT]

    image_dir_ORB = [x for x in image_dir if x not in
        remove_ORB]

    class_list_SIFT = random.sample(image_dir_SIFT, num)
        #Random selection of classes picked out

    class_list_ORB = random.sample(image_dir_ORB, num)

    return class_list_SIFT, class_list_ORB

def train_test(path, num):

    class_list = get_classes(path, num) #Retrieves the
        random classes
    print(f"{len(class_list[0])} classes were returned:\n
        {class_list[0]} for SIFT")

    train_list = [[] for x in range(len(class_list[0]))]
        #Creates a list of lists where each nested list
        corresponds to a class
    test_list = [[] for x in range(len(class_list[0]))]

    train_labels = np.zeros(1)
    test_labels = np.zeros(1)

    for i, cl in enumerate (class_list[0]): #Iterate
        through each class

        class_path = path + cl + '/images/'
        image_dir = os.listdir(class_path) #Path
            directory to images within the class
        train_dir = image_dir[0:400] #First 400 images
            added to training set and last 100 to test set
        test_dir = image_dir[400:]

        #Iterate through all images and create a full
            path directory to that image and add to list
        #All images can then be easily accessed
        for j, train in enumerate (train_dir):
            img_path = os.path.join(class_path, train)
            train_dir[j] = img_path
```

```
        for k, test in enumerate (test_dir):
            img_path = os.path.join(class_path, test)
            test_dir[k] = img_path

        train_list[i] = train_dir
        test_list[i] = test_dir

        train_labels = np.append(train_labels, np.full(
            len(train_dir), i))
        test_labels = np.append(test_labels, np.full(len(
            test_dir), i))

    train_labels = train_labels[1:]
    test_labels = test_labels[1:]

    return train_list, test_list, train_labels,
        test_labels

#Seperate train test function for ORB as the classes used
     differ from SIFT
def train_test_ORB(path, num):

    class_list = get_classes(path, num) #Retrieves the
        random classes

    class_list_actual = ['n01742172', 'n03160309', '
        n02509815', 'n02802426', 'n01944390', 'n02395406',
         'n01917289', 'n01784675', 'n02814533', 'n02190166
        ', 'n02927161', 'n02917067', 'n02481823', '
        n01882714', 'n02950826']
    print(f"{len(class_list[1])} classes were returned:\n
        {class_list[1]} for ORB")

    train_list = [[] for x in range(len(class_list[1]))]
        #Creates a list of lists where each nested list
        corresponds to a class
    test_list = [[] for x in range(len(class_list[1]))]

    train_labels = np.zeros(1)
    test_labels = np.zeros(1)

    for i, cl in enumerate (class_list[1]): #Iterate
        through each class

        class_path = path + cl + '/images/'
```

```
        image_dir = os.listdir(class_path) #Path
            directory to images within the class
        train_dir = image_dir[0:400] #First 400 images
            added to training set and last 100 to test set
        test_dir = image_dir[400:]

        #Iterate through all images and create a full
            path directory to that image and add to list
        #All images can then be easily accessed
        for j, train in enumerate (train_dir):
            img_path = os.path.join(class_path, train)
            train_dir[j] = img_path

        for k, test in enumerate (test_dir):
            img_path = os.path.join(class_path, test)
            test_dir[k] = img_path

        train_list[i] = train_dir
        test_list[i] = test_dir

        train_labels = np.append(train_labels, np.full(
            len(train_dir), i))
        test_labels = np.append(test_labels, np.full(len(
            test_dir), i))

    train_labels = train_labels[1:]
    test_labels = test_labels[1:]

    return train_list, test_list, train_labels,
        test_labels

random.seed(42)

image_path = 'TinyImageNet100/TinyImageNet100/'
num_classes = 15

#Train and test image lists and labels obtained for SIFT
    model
train_list, test_list, train_labels, test_labels =
    train_test(image_path, num_classes)
#Train and test image lists and labels obtained for ORB
    model
train_list_ORB, test_list_ORB, train_labels_ORB,
    test_labels_ORB = train_test_ORB(image_path,
    num_classes)
```

```
train_list[0]

def GetClusterCentres_SIFT(img_paths,num_words):

    num_classes = len(img_paths)

    sift_des = SIFT() #Using the SIFT model
    des_list = [[] for x in range(num_classes)] #
        Descriptor list where each index will contain all
        descriptors for given image
    key_list = [[] for x in range(num_classes)]
    des_matrix = [np.zeros((1,128)) for x in range(
        num_classes)] #Will contain all descriptors for
        all images stacked on top of each other

    for i in range(num_classes): #Iterate through all the
         classes

        for paths in img_paths[i]:

            img = cv2.imread(paths) #Read in the image
                from img_paths
            img_gray = cv2.cvtColor(img, cv2.
                COLOR_RGB2GRAY) #Turn image gray
            sift_des.detect_and_extract(img_gray) #
                Extract the keypoints and descriptors
            keypoints = sift_des.keypoints
            descriptors = sift_des.descriptors
            #If the descriptor around does contain values
                 then add to des_matrix
            if descriptors is not None:
                des_matrix[i] = np.row_stack((des_matrix[
                    i], descriptors))
            #Append descriptor to list
            des_list[i].append(descriptors)
            key_list[i].append(keypoints)

    all_centres = [[] for x in range(num_classes)]

    #Ignore first row as it only contains zeros
    for i in range(num_classes):
        des_matrix[i] = des_matrix[i][1:, :]

        kmeans = KMeans(n_clusters=num_words,
            random_state=33)
        kmeans.fit(des_matrix[i])
```

```
        centres = kmeans.cluster_centers_
        all_centres[i] = centres

    return all_centres,des_list, key_list

def GetClusterCentres_ORB(img_paths,num_words):

    num_classes = len(img_paths)

    orb_des = ORB() #Using the ORB model
    des_list = [[] for x in range(num_classes)] #
       Descriptor list where each index will contain all
       descriptors for given image
    des_matrix = [np.zeros((1,256)) for x in range(
       num_classes)] #Will contain all descriptors for
       all images stacked on top of each other

    for i in range(num_classes): #Iterate through all the
        classes

        for paths in img_paths[i]:

            img = cv2.imread(paths) #Read in the image
               from img_paths
            img = cv2.resize(img, (80,80))
            img_gray = cv2.cvtColor(img, cv2.
               COLOR_RGB2GRAY) #Turn image gray
            orb_des.detect_and_extract(img_gray) #Extract
                the keypoints and descriptors
            keypoints = orb_des.keypoints
            descriptors = (orb_des.descriptors).astype(np
               .float32)

            #If the descriptor around does contain values
                then add to des_matrix
            if descriptors is not None:
                des_matrix[i] = np.row_stack((des_matrix[
                   i], descriptors))
            #Append descriptor to list
            des_list[i].append(descriptors)

    all_centres = [[] for x in range(num_classes)]

    #Ignore first row as it only contains zeros
    for i in range(num_classes):
        des_matrix[i] = des_matrix[i][1:, :]
```

```
        kmeans = KMeans(n_clusters=num_words,
            random_state=33)
        kmeans.fit(des_matrix[i])
        centres = kmeans.cluster_centers_
        all_centres[i] = centres

    return all_centres,des_list

#Number of words to describe each image
num_words = 10

centres_SIFT, des_list_train_SIFT, key_list_train_SIFT =
    GetClusterCentres_SIFT(train_list, num_words =
    num_words)
centres_SIFT, des_list_test_SIFT, key_list_test_SIFT =
    GetClusterCentres_SIFT(test_list, num_words =
    num_words)

centres_ORB, des_list_train_ORB = GetClusterCentres_ORB(
    train_list_ORB, num_words = num_words)
centres_ORB, des_list_test_ORB = GetClusterCentres_ORB(
    test_list_ORB, num_words = num_words)

#Function is applied to each image through each iteration
def des2feature_SIFT(des,num_words,centures):

    #The 10 words to describe the image (histogram). Ok,
        so not actuals words, each index corresponds to
        particular feature.
    img_feature_vec=np.zeros((1,num_words),'float32')

    #des.shape[0] is the number of descriptor arrays for
        given image. So, iterating through each layer of
        descriptor
    for i in range(des.shape[0]):
        feature_k_rows=np.ones((num_words,128),'float32')
            #Matrix created to align with number of
            centres (10
        feature=des[i] #ith layer of descriptor
        feature_k_rows=feature_k_rows*feature
        feature_k_rows=np.sum((feature_k_rows-centures)
            **2,1) #computes the squared difference
            between the descriptor and each cluster center
            . Then compute sum of squared differences
```

```
        index=np.argmin(feature_k_rows) #Finds index of
            cluster centre closest to descriptor
        img_feature_vec[0][index]+=1 #Increment count of
            corresponding cluster in histogram

    return img_feature_vec

#Function is applied to each image through each iteration
def des2feature_ORB(des,num_words,centures):

    #The 10 words to describe the image (histogram). Ok,
        so not actuals words, each index corresponds to
        particular feature.
    img_feature_vec=np.zeros((1,num_words),'float32')

    #des.shape[0] is the number of descriptor arrays for
        given image. So, iterating through each layer of
        descriptor
    for i in range(des.shape[0]):
        feature_k_rows=np.ones((num_words,256),'float32')
            #Matrix created to align with number of
            centres (10
        feature=des[i] #ith layer of descriptor
        feature_k_rows=feature_k_rows*feature
        feature_k_rows=np.sum((feature_k_rows-centures)
            **2,1) #computes the squared difference
            between the descriptor and each cluster center
            . Then compute sum of squared differences


        index=np.argmin(feature_k_rows) #Finds index of
            cluster centre closest to descriptor
        img_feature_vec[0][index]+=1 #Increment count of
            corresponding cluster in histogram

    return img_feature_vec

def get_all_features_SIFT(des_list,num_words, centres):

    #This will contain all image feature vectors. 21 rows
        (because 21 images) and 10 columns
    allvec= [(np.zeros((len(des_list[0]),num_words),'
        float32')) for x in range(len(des_list))]

    for i in range(len(des_list)):
```

```
        #Iterate through each image
        for idx, j in enumerate (des_list[i]):

            if j.any()!=None:
                #Each iteration adds set of 10 words for
                    each image.
                allvec[i][idx]=des2feature_SIFT(centures=
                    centres[i],des=j,num_words=num_words)

    return allvec

def get_all_features_ORB(des_list,num_words, centres):

    #This will contain all image feature vectors. 21 rows
        (because 21 images) and 10 columns
    allvec= [(np.zeros((len(des_list[0]),num_words),'
       float32')) for x in range(len(des_list))]

    for i in range(len(des_list)):

        #Iterate through each image
        for idx, j in enumerate (des_list[i]):
            if j is not None and j.size > 0:
                #Each iteration adds set of 10 words for
                    each image.
                allvec[i][idx]=des2feature_ORB(centures=
                    centres[i],des=j,num_words=num_words)

    return allvec

def fisher_vec(train_descriptors, test_descriptors,
   train_targets, test_targets):

    b = 0
    prediction_arr = []
    for i in range(len(train_descriptors)):
        print(i)
        k = 16
        gmm = learn_gmm(train_descriptors[i], n_modes=k)

        training_fvs = np.array([fisher_vector(
           descriptor_mat, gmm) for descriptor_mat in
           train_descriptors[i]])

        test_fvs = np.array([fisher_vector(descriptor_mat
           , gmm) for descriptor_mat in test_descriptors[
```

```
            i]])

        svm = LinearSVC().fit(training_fvs, train_targets
            [b: b + 400])

        predictions = svm.predict(testing_fvs)

        prediction_arr.append(predictions)

        b += 400

    print(prediction_arr)
    print(classification_report(test_targets, predictions
        ))

    ConfusionMatrixDisplay.from_estimator(
        svm,
        testing_fvs,
        test_targets,
        cmap=plt.cm.Blues,
    )

    plt.show()

img_features_train_SIFT = get_all_features_SIFT(des_list=
    des_list_train_SIFT,num_words=num_words, centres =
    centres_SIFT)
img_features_test_SIFT = get_all_features_SIFT(des_list=
    des_list_test_SIFT,num_words=num_words, centres =
    centres_SIFT)

#Feature vectors for each image concatenated into one
    array to apply Linear SVC
img_features_train_SIFT = np.concatenate(np.array(
    img_features_train_SIFT), axis = 0)
img_features_test_SIFT = np.concatenate(np.array(
    img_features_test_SIFT), axis = 0)

img_features_train_ORB =get_all_features_ORB(des_list=
    des_list_train_ORB,num_words=num_words, centres =
    centres_ORB)
img_features_test_ORB =get_all_features_ORB(des_list=
    des_list_test_ORB,num_words=num_words, centres =
    centres_ORB)
```

```
img_features_train_ORB = np.concatenate(np.array(
    img_features_train_ORB), axis = 0)
img_features_test_ORB = np.concatenate(np.array(
    img_features_test_ORB), axis = 0)

#Applying Support Vector Machine to feature vectors
    obtained using SIFT
svm_SIFT = LinearSVC().fit(img_features_train_SIFT,
    train_labels)
#Predict target labels from test images
predictions_SIFT = svm_SIFT.predict(
    img_features_test_SIFT)

print(f"CLASSIFICATION REPORT:\n{num_classes} class image
     classification using SIFT\n" + classification_report(
    test_labels, predictions_SIFT))
#Applying Support Vector Machine to feature vectors
    obtained using ORB
svm_ORB = LinearSVC().fit(img_features_train_ORB,
    train_labels_ORB)
#Predict target labels from test images
predictions_ORB = svm_ORB.predict(img_features_test_ORB)

print(f"CLASSIFICATION REPORT:\n{num_classes} class image
     classification using ORB\n" + classification_report(
    test_labels, predictions_ORB))

fig, ax = plt.subplots(1,2, figsize = (12,6))

ConfusionMatrixDisplay.from_estimator(
    svm_SIFT,
    img_features_test_SIFT,
    test_labels,
    cmap=plt.cm.Blues,
    ax=ax[0]
)

ax[0].set_title(f"{num_classes} classes image
    classification using SIFT")
ax[0].set_xticklabels(range(15),rotation = 45)

ConfusionMatrixDisplay.from_estimator(
    svm_ORB,
    img_features_test_ORB,
    test_labels_ORB,
    cmap=plt.cm.Blues,
```

```
    ax=ax[1]
)

ax[1].set_title(f"{num_classes} classes image
    classification using ORB")
ax[1].set_xticklabels(range(15),rotation = 45)
plt.tight_layout()
plt.show()

#fisher_vec(des_list_train_ORB, des_list_test_ORB,
    train_labels_ORB, test_labels_ORB)

def NN_data_preprocess_train(data, data_labels, batch):

    #One transform for transforming image to tensor and
        one for normalizing
    transform_ToTen = transforms.ToTensor()
    transform_Norm = transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224, 0.225])

    data_list = [] #List for appending accepted images
    labels_list = [] #List for appending the
        corressponding labels

    data = np.array(data).flatten().tolist() #Flattening
        training list so all paths can be iterated through
         at once
    data_labels = data_labels.tolist()

    for idx, path in enumerate(data):
        #image loaded using PIL and transformed to tensor
        image = Image.open(path)
        image = transform_ToTen(image)

        if image.shape[0] == 3: #Check that image is RBG.
             Shape could only be checked for non-
            normalized image
            #Image normalized and added to list along
                with label
            image = transform_Norm(image)
            data_list.append(image)
            labels_list.append(data_labels[idx])

    data_tensor = torch.stack(data_list)
    labels_tensor = torch.tensor(labels_list).long()
```

```
    dataset = TensorDataset(data_tensor, labels_tensor) #
        images and labels combined into one dataset for
        easier examination in NN

    dataloader = DataLoader(dataset, batch_size=batch,
        shuffle=True) #Data split into batches of size 32

    return dataloader

def NN_data_preprocess_test(data, data_labels, batch):

    #One transform for transforming image to tensor and
        one for normalizing
    transform_ToTen = transforms.ToTensor()
    transform_Norm = transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224, 0.225])

    data_list = [] #List for appending accepted images
    labels_list = [] #List for appending the
        corressponding labels

    data = np.array(data).flatten().tolist() #Flattening
        test list so all paths can be iterated through at
        once
    data_labels = data_labels.tolist()

    for idx, path in enumerate(data):
        #image loaded using PIL and transformed to tensor
        image = Image.open(path)
        image = transform_ToTen(image)

        if image.shape[0] == 3: #Check that image is RBG.
            Shape could only be checked for non-
            normalized image
            #Image normalized and added to list along
                with label
            image = transform_Norm(image)
            data_list.append(image)
            labels_list.append(data_labels[idx])

    data_tensor = torch.stack(data_list)
    labels_tensor = torch.tensor(labels_list).long()

    dataset = TensorDataset(data_tensor, labels_tensor) #
        images and labels combined into one dataset for
        easier examination in NN
```

```
    dataloader = DataLoader(dataset, batch_size=batch,
        shuffle=False) #Data split into batches of size 32

    return dataloader

train_dataloader = NN_data_preprocess_train(train_list,
    train_labels, 32)
test_dataloader = NN_data_preprocess_test(test_list,
    test_labels, 32)

train_features_batch, train_labels_batch = next(iter(
    train_dataloader))
train_features_batch.shape, train_labels_batch.shape

#Show a random image from a random class
torch.manual_seed(42)
random_idx = torch.randint(0, len(train_features_batch),
    size=[1]).item()
img, label = train_features_batch[random_idx],
    train_labels_batch[random_idx]
img = img.permute(1,2,0)
plt.imshow(img.squeeze())
plt.title(label)
plt.axis("Off");
print(f"Image size: {img.shape}")
print(f"Label: {label}, label size: {label.shape}")

#Implements a progress bar
def print_train_time(start: float, end: float, device:
    torch.device = None):
    total_time = end - start
    print(f"Train time on {device}: {total_time:.3f}
        seconds\n")
    return total_time

device = "cuda" if torch.cuda.is_available() else "cpu"

def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               accuracy_fn,
               epoch, train_loss_list, train_acc_list,
               device: torch.device = device):
```

```
#Only print updates every 10 epochs
if epoch % 10 == 0:
    print(f"Training Epoch: {epoch}")

train_loss, train_acc = 0, 0
model.to(device)

for batch, (X, y) in enumerate(data_loader):
    # Send data to GPU
    X, y = X.to(device), y.to(device)

    model.train()

    #Forward pass to make y prediction
    y_pred = model(X)

    #Calculate loss between predicted and true y
        value
    loss = loss_fn(y_pred, y)
    train_loss += loss
    train_acc += accuracy_fn(y_true=y,
                             y_pred=y_pred.argmax(dim
                                 =1)) # Go from logits
                                  -> pred labels

    #Resets the gradients to zero
    optimizer.zero_grad()

    #Backward pass to calculate gradients
    loss.backward()

    #Optimizer step to adjust weights according to
        gradient
    optimizer.step()

    # Print out how many samples have been seen
    if epoch % 10 == 0:
        if batch % 30 == 0:
            print(f"Looked at {batch * len(X)}/{len(
                data_loader.dataset)} samples")

# Divide total train loss by length of train
    dataloader (average loss per batch per epoch)
train_loss /= len(data_loader)
train_acc /= len(data_loader)
```

```
    #Add loss and accuracy to list to track them after
        each epoch
    train_loss_list.append(train_loss.item())
    train_acc_list.append(train_acc)

    if epoch % 10 == 0:
        print(f"\nTrain Loss: {train_loss:.5f}\n")

def test_step(data_loader: torch.utils.data.DataLoader,
              model: torch.nn.Module,
              loss_fn: torch.nn.Module,
              accuracy_fn,
              epoch, test_loss_list, test_acc_list,
                  early_stop_patience, best_val_loss,
                  early_stop_counter,
              device: torch.device = device):
    test_loss, test_acc = 0, 0
    model.to(device)
    model.eval() # put model in eval mode
    # Turn on inference context manager
    with torch.inference_mode():
        for X, y in data_loader:
            # Send data to GPU
            X, y = X.to(device), y.to(device)

            #Forward pass
            test_pred = model(X)

            #Calculate loss and accuracy
            test_loss += loss_fn(test_pred, y)
            test_acc += accuracy_fn(y_true=y,
                y_pred=test_pred.argmax(dim=1) # Go from
                    logits -> pred labels
            )

        # Adjust metrics and print out
        test_loss /= len(data_loader)
        test_acc /= len(data_loader)

        test_loss_list.append(test_loss.item())
        test_acc_list.append(test_acc)

        if epoch % 10 == 0:
            print(f"\nTest Loss: {test_loss:.5f}\n")

torch.manual_seed(42)
```

```
class NN_Classification(nn.Module):
    def __init__(self, input_shape: int, hidden_units:
        int, output_shape: int):
        super().__init__()
        self.layer_stack = nn.Sequential(
            nn.Flatten(), # neural networks like their
                inputs in vector form
            nn.Dropout(p = 0.5), #50% probability of node
                 being zeroed to improve models ability to
                 generalise
            nn.Linear(in_features=input_shape,
                out_features=hidden_units), # in_features
                = number of features in a data sample (784
                 pixels)
            nn.ReLU(),
            nn.Linear(in_features=hidden_units,
                out_features=hidden_units),
            nn.ReLU(),
            nn.Linear(in_features=hidden_units,
                out_features=output_shape),
            nn.Softmax(dim=1) #Softmax activation
                function to introduce non-linearity
        )

    def forward(self, x):
        return self.layer_stack(x)

# Need to setup model with input parameters
model_0 = NN_Classification(input_shape=12288, # one for
    every pixel
    hidden_units=128, # how many units in the hidden
        layer
    output_shape=num_classes # one for every class
)
model_0.to("cpu") # keep model on CPU to begin with

# Setup loss function and optimizer
import torch.optim.lr_scheduler as lr_scheduler
loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(params=model_0.parameters(),
    lr = 0.001, weight_decay=1e-4)
#scheduler = lr_scheduler.StepLR(optimizer, step_size=10,
    gamma = 0.1)

# Import tqdm for progress bar
```

```
from tqdm.auto import tqdm
from helper_functions import accuracy_fn

torch.manual_seed(42)

# Measure time
from timeit import default_timer as timer
train_time_start_model_2 = timer()

# Train and test model
epochs = 150
loss_list=[]

train_loss_list = []
train_acc_list = []

test_loss_list = []
test_acc_list = []

#Patience level to determine when model stops learning
    depending how far validation strays from training
    accuracy
early_stop_patience = 1
best_val_loss = float('inf')
early_stop_counter = 0

for epoch in tqdm(range(epochs)):

    train_step(data_loader=train_dataloader,
        model=model_0,
        loss_fn=loss_func,
        optimizer=optimizer,
        accuracy_fn=accuracy_fn,
        device=device,
        epoch=epoch,
        train_loss_list = train_loss_list,
        train_acc_list = train_acc_list
    )
    test_step(data_loader=test_dataloader,
        model=model_0,
        loss_fn=loss_func,
        accuracy_fn=accuracy_fn,
        device=device,
        epoch=epoch,
        test_loss_list = test_loss_list,
        test_acc_list = test_acc_list,
```

```
        early_stop_patience = early_stop_patience,
        best_val_loss = best_val_loss,
        early_stop_counter = early_stop_counter
    )

    if test_loss_list[epoch] < best_val_loss:
        best_val_loss = test_loss_list[epoch]
        early_stop_counter = 0  # Reset the counter if
            validation loss improves
    else:
        early_stop_counter += 1  # Increment the counter
            if no improvement

    # Check for early stopping
    if early_stop_counter >= early_stop_patience:
        print(f"Early stopping triggered at epoch {epoch
            }")
        break

train_time_end_model_2 = timer()
total_train_time_model_2 = print_train_time(start=
    train_time_start_model_2,
                                            end=
                                                train_time_end_model_2
                                                ,
                                            device=device)

print(f"Training Accuracy: {train_acc_list[-1]}")
print(f"Test Accuracy: {test_acc_list[-1]}")

print(f"Training Loss: {train_loss_list[-1]}")
print(f"Test Loss: {test_loss_list[-1]}")

plt.plot(np.arange(len(train_acc_list)), train_acc_list,
    label = 'Training')
plt.plot(np.arange(len(test_acc_list)), test_acc_list,'r
    ', label = 'Test')
plt.xlabel("Epoch number")
plt.ylabel("Accuracy / %")
plt.legend()
plt.show()

torch.manual_seed(42)
def eval_model(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
```

```
                accuracy_fn):
    loss, acc = 0, 0
    acc_list = []
    model.eval()
    with torch.inference_mode():
        for X, y in data_loader:

            y = y.long()

            # Make predictions with the model
            y_pred = model(X)

            # Accumulate the loss and accuracy values per
                batch
            loss += loss_fn(y_pred, y)
            acc += accuracy_fn(y_true=y,
                               y_pred=y_pred.argmax(dim
                                   =1)) # For accuracy,
                                   need the prediction
                                   labels (logits ->
                                   pred_prob ->
                                   pred_labels)
            acc_list.append(acc)

        # Scale loss and acc to find the average loss/acc
            per batch
        loss /= len(data_loader)
        acc /= len(data_loader)

        acc_list = np.array(acc_list)
        acc_list /= len(data_loader)

    return (model.__class__.__name__, loss.item(), acc,
        acc_list)

# Calculate model 0 results on test dataset
model_0_results = eval_model(model=model_0, data_loader=
    test_dataloader,
                             loss_fn=loss_func,
                                 accuracy_fn=accuracy_fn)

print(f"Model Name: {model_0_results[0]}\nloss_fn: {
    model_0_results[1]}\nAccuracy: {model_0_results[2]}")

plt.plot(np.arange(0,len(model_0_results[3])),
    model_0_results[3])
```

```
plt.show()

def CNN_data_preprocess_train(data, data_labels, batch):

    #One transform for transforming image to tensor and
        one for normalizing
    transform_ToTen = transforms.ToTensor()
    transform_Norm = transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224, 0.225])

    train_transform = transforms.Compose([
        transforms.RandomRotation(15),
        transforms.RandomAffine(degrees=15, translate
            =(0.1, 0.1), scale=(0.9, 1.1), shear=10),
        transforms.RandomHorizontalFlip(),
        transforms.ColorJitter(brightness=0.1, contrast
            =0.1, saturation=0.1, hue=0.05),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225])
    ])

    data_list = [] #List for appending accepted images
    labels_list = [] #List for appending the
        corressponding labels

    data = np.array(data).flatten().tolist() #Flattening
        training list so all paths can be iterated through
         at once
    data_labels = data_labels.tolist()

    for idx, path in enumerate(data):
        #image loaded using PIL and transformed to tensor
        image = Image.open(path)
        image = transform_ToTen(image)

        if image.shape[0] == 3: #Check that image is RBG.
             Shape could only be checked for non-
            normalized image
            #Transforms applied to images and added to
                list along with label
            image = train_transform(image)
            data_list.append(image)
            labels_list.append(data_labels[idx])

    data_tensor = torch.stack(data_list)
    labels_tensor = torch.tensor(labels_list).long()
```

```
    dataset = TensorDataset(data_tensor, labels_tensor) #
        images and labels combined into one dataset for
        easier examination in NN

    dataloader = DataLoader(dataset, batch_size=batch,
        shuffle=True) #Data split into batches of size 32

    return dataloader

def CNN_data_preprocess_test(data, data_labels, batch):

    #One transform for transforming image to tensor and
        one for normalizing
    transform_ToTen = transforms.ToTensor()
    transform_Norm = transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224, 0.225])

    data_list = [] #List for appending accepted images
    labels_list = [] #List for appending the
        corressponding labels

    data = np.array(data).flatten().tolist() #Flattening
        test list so all paths can be iterated through at
        once
    data_labels = data_labels.tolist()

    for idx, path in enumerate(data):
        #image loaded using PIL and transformed to tensor
        image = Image.open(path)
        image = transform_ToTen(image)

        if image.shape[0] == 3: #Check that image is RBG.
            Shape could only be checked for non-
            normalized image
            #Image normalized and added to list along
                with label
            image = transform_Norm(image)
            data_list.append(image)
            labels_list.append(data_labels[idx])

    data_tensor = torch.stack(data_list)
    labels_tensor = torch.tensor(labels_list).long()

    dataset = TensorDataset(data_tensor, labels_tensor) #
        images and labels combined into one dataset for
```

```
        easier examination in NN

    dataloader = DataLoader(dataset, batch_size=batch,
        shuffle=False) #Data split into batches of size 32

    return dataloader

train_dataloader_conv = CNN_data_preprocess_train(
    train_list, train_labels, 32)
test_dataloader_conv = CNN_data_preprocess_test(test_list
    , test_labels, 32)

# Create a convolutional neural network
device = "cuda" if torch.cuda.is_available() else "cpu"
class Conv_NN(nn.Module):
    def __init__(self, input_shape: int, hidden_units:
        int, output_shape: int):
        super().__init__()
        self.block_1 = nn.Sequential(
            nn.Conv2d(in_channels=input_shape,
                      out_channels=hidden_units, #Takes
                          the input image and applies
                          hidden filters to it
                      kernel_size=3, # how big is the
                          square that's going over the
                          image?
                      stride=1, # default
                      padding=1),# options = "valid" (no
                          padding) or "same" (output has
                          same shape as input) or int for
                          specific number
            nn.BatchNorm2d(hidden_units),
            nn.ReLU(), #Apply non-linearity so model can
                learn complex patterns
            nn.Conv2d(in_channels=hidden_units, #This
                second layer applies another layer of
                hidden filters to detect edges and
                features
                      out_channels=hidden_units,
                      kernel_size=3,
                      stride=1,
                      padding=1),
            nn.BatchNorm2d(hidden_units),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, #Reduces spatial
                dimension by half by taking max value in a
```

```
                2x2 block
                        stride=2) # default stride value
                            is same as kernel_size
        )
        self.block_2 = nn.Sequential(
            nn.Conv2d(hidden_units, hidden_units, 3,
                padding=1),
            nn.ReLU(),
            nn.Conv2d(hidden_units, hidden_units, 3,
                padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        #Dummy tensor to compute final feature map size
        #This was added so that if anymore blocks were
            added that alter the image size then the
            classifier will dynamically know what image
            size to expect
        with torch.no_grad():
            dummy_input = torch.randn(1, 3, 64, 64)
            out = self.block_1(dummy_input)
            out = self.block_2(out)
            self.flattened_size = out.shape[1] * out.
                shape[2] * out.shape[3]  # (channels *
                height * width)

        self.classifier = nn.Sequential(
            nn.Flatten(), #Convert to 1D tensor
            nn.Linear(self.flattened_size, 256),
            nn.ReLU(),
            nn.Linear(256, output_shape)
        )

    def forward(self, x: torch.Tensor):
        x = self.block_1(x)
        #print(x.shape) #Import print statements to
            determine the shape of the image as it passes
            through layers
        x = self.block_2(x)
        #print(x.shape)

        x = self.classifier(x)
        #print(x.shape)
        return x
```

```
torch.manual_seed(42)
model_CNN = Conv_NN(input_shape=3,
    hidden_units=128,
    output_shape=num_classes).to(device)
model_CNN

loss_func_conv = nn.CrossEntropyLoss()
optimizer_conv = torch.optim.SGD(params=model_CNN.
   parameters(), lr = 0.001)

torch.manual_seed(42)

# Measure time
train_time_start_model_CNN = timer()

# Train and test model
epochs = 15
loss_list=[]

train_loss_list = []
train_acc_list = []

test_loss_list = []
test_acc_list = []

early_stop_patience = 5
best_val_loss = float('inf')
early_stop_counter = 0

for epoch in tqdm(range(epochs)):

    train_step(data_loader=train_dataloader_conv,
        model=model_CNN,
        loss_fn=loss_func_conv,
        optimizer=optimizer_conv,
        accuracy_fn=accuracy_fn,
        device=device,
        epoch=epoch,
        train_loss_list = train_loss_list,
        train_acc_list = train_acc_list
    )
    test_step(data_loader=test_dataloader_conv,
        model=model_CNN,
        loss_fn=loss_func_conv,
        accuracy_fn=accuracy_fn,
        device=device,
```

```
        epoch=epoch,
        test_loss_list = test_loss_list,
        test_acc_list = test_acc_list,
        early_stop_patience = early_stop_patience,
        best_val_loss = best_val_loss,
        early_stop_counter = early_stop_counter
    )

    if test_loss_list[epoch] < best_val_loss:
        best_val_loss = test_loss_list[epoch]
        early_stop_counter = 0  # Reset the counter if
            validation loss improves
    else:
        early_stop_counter += 1  # Increment the counter
            if no improvement

    # Check for early stopping
    if early_stop_counter >= early_stop_patience:
        print(f"Early stopping triggered at epoch {epoch
            }")
        break

train_time_end_model_CNN = timer()
total_train_time_model_CNN = print_train_time(start=
    train_time_start_model_CNN,
                                              end=
                                                  train_time_end_model_CNN
                                                  ,
                                              device=device)

print(f"Training Accuracy: {train_acc_list[-1]}")
print(f"Test Accuracy: {test_acc_list[-1]}")

print(f"Training Loss: {train_loss_list[-1]}")
print(f"Test Loss: {test_loss_list[-1]}")

plt.plot(np.arange(len(train_acc_list)), train_acc_list,
    label = 'Training')
plt.plot(np.arange(len(test_acc_list)), test_acc_list,'r
    ', label = 'Test')
plt.xlabel("Epoch number")
plt.ylabel("Accuracy / %")
plt.legend()
plt.show()

# Get model_2 results
```

```
torch.manual_seed(42)
def eval_model(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               accuracy_fn):
    """Returns a dictionary containing the results of
       model predicting on data_loader.

    Args:
        model (torch.nn.Module): A PyTorch model capable
            of making predictions on data_loader.
        data_loader (torch.utils.data.DataLoader): The
            target dataset to predict on.
        loss_fn (torch.nn.Module): The loss function of
            model.
        accuracy_fn: An accuracy function to compare the
            models predictions to the truth labels.

    Returns:
        (dict): Results of model making predictions on
            data_loader.
    """
    loss, acc = 0, 0
    model.eval()
    with torch.inference_mode():
        for X, y in data_loader:
            # Make predictions with the model
            y_pred = model(X)

            # Accumulate the loss and accuracy values per
                batch
            loss += loss_fn(y_pred, y)
            acc += accuracy_fn(y_true=y,
                               y_pred=y_pred.argmax(dim
                                   =1)) # For accuracy,
                                   need the prediction
                                   labels (logits ->
                                   pred_prob ->
                                   pred_labels)

        # Scale loss and acc to find the average loss/acc
            per batch
        loss /= len(data_loader)
        acc /= len(data_loader)
```

```
    return {"model_name": model.__class__.__name__, #
        only works when model was created with a class
            "model_loss": loss.item(),
            "model_acc": acc}

model_CNN_results = eval_model(
    model=model_CNN,
    data_loader=test_dataloader,
    loss_fn=loss_func,
    accuracy_fn=accuracy_fn
)
model_CNN_results
```